On the Performance of Network Parallel Training in Artificial Neural Networks

Ludvig Ericson, Rendani Mbuvha
School of Computer Science & Communication
KTH Royal Institute of Technology
Stockholm, Sweden
{ludv,rendani}@kth.se

Bhekisipho Twala
Institute of Intelligent Systems
University of Johannesburg
Johannesburg, South Africa
btwala@uj.ac.za

***Abstract*—Artificial Neural Networks (ANNs) have received increasing attention in recent years with applications that span a wide range of disciplines including vital domains such as medicine, network security and autonomous transportation. However, neural network architectures are becoming increasingly complex and with an increasing need to obtain real-time results from such models, it has become pivotal to use parallelization as a mechanism for speeding up network training and deployment. In this work we propose an implementation of Network Parallel Training through Cannon's Algorithm for matrix multiplication. We show that increasing the number of processes speeds up training until the point where process communication costs become prohibitive; this point varies by network complexity. We also show through empirical efficiency calculations that the speedup obtained is superlinear.**

I. Introduction

Artificial neural networks, or ANNs, are computational tools that are inspired by their biological namesake. ANNs are particularly useful when estimating functions of arbitrary complexity using given example data[1]. This inturn makes them well suited for pattern recognition tasks where they have been successfully applied in dimensionality reduction, time series prediction, and data mining. While they have long been on the fringes of the research community since their inception in the '80s, recent advances in computer and material sciences have allowed for an unprecedented growth in computation power, and thus allowing the possibility of even more sophisticated networks. One of the drawbacks of modern ANNs is the time it takes to train such a network.

As society becomes increasingly dependent on machine learning applications, real-time output from on-line training of neural network systems is critical to positive user experiences and usefulness of such systems. Thus speeding up neural network training with a parallel algorithm as we propose has become essential to the efficient deployment of machine learning applications. The case for parallelization is also reinforced by the increase in availability of multicore CPUs and general purpose graphical processing units in recent years.

Most early work in parallelization of neural networks like [2] primarily suggest special customised hardware. This approach has certain limitations as such hardware is costly and may display suboptimal performance when utilized for other tasks. Consequently, this can be impractical for modern scalability platforms such as HPC clusters and cloud computing infrastructure.

Dahl et al. [3] present a Pattern Parallel Training (PPT) algorithm for parallel ANN training. In PPT, multiple ANNs are duplicated in different processes. In each process an ANN is trained on a randomly selected subset of the training examples. At the end of an epoch each process broadcasts its weight updates which are then applied to all other ANNs. This has the effect of reducing the training time by faster convergence.

Suri et al. [4] use a similar approach to [3], however, at the end of each full training round they apply an evolutionary algorithm to select the best candidate networks for further training.

The approach presented here is most similar to our work is [5] who also use a Node Parallelization strategy, while communicating the weights corresponding to "ghost" neurons. Their strategy requires that each process is fed with all the inputs. It therefore follows that this becomes onerous on the amount of memory required.

The contribution and novelty in our proposed approach centers on its memory efficiency in that by using Cannon's algorithm it does not require all the inputs to be fed to each process and that no specialized hardware is required. The remainder of this paper is organised as follows: Section II describes Artificial Neural Networks, Backpropagation and our proposed implementation of Network Parallel Training. Section III sets out our experiments. Section IV discusses the experimental results. We conclude our paper in section V

II. Methods

A. Artificial Neural Networks

An ANN is defined by a weighted, directed graph $G = (V, E)$, where the vertices are likened to neurons; edges to "synapses," the connections between neurons. The weights are sensitivities, e.g. a highly-weighted connection will contribute more to the excitation of a neuron. We will concern ourselves only with fully-connected feed-forward networks. A feed-forward network is one in which the graph is acyclic, and so the neurons can be subdivided into layers. Being fully-connected, all layers are maximally connected to the next. The output z_j of some neuron j is thus a function of the output of the previous layer. We define it as

$$z_j = f_j(\mathbf{x}) = h(\sum_{\{i,j\}\in E} w_{ij} z_i + b_j),$$

where h is some non-linear activation function, g_i is the output of the ith neuron in the previous layer, w_{ij} is the weight for that connection, and b_j is the neuron's bias. A choice for the non-linearity function h that has gained popularity in recent years is the ReLU, or *rectified linear unit*. We will use the leaky ReLU, defined as

$$h(x) = \begin{cases} x & \text{if } x > 0, \\ 0.01x & \text{otherwise.} \end{cases}$$

The activation can be described in terms of a matrix multiplication. Let W_i be the weights of ith layer, with each neuron as a column of weights. The activations Z_i for that layer on a row-wise subset X of the training data is then

$$Z_i = h(XW_i + \mathbf{b}_i).$$

B. Training Neural Networks

Training a neural network is done with *stochastic gradient descent*, or SGD. In SGD, the gradient of the loss function L is computed for a single sample. The network parameters are then updated using the expression $\boldsymbol{\theta}' = \boldsymbol{\theta} - \eta\nabla L$, where $0 < \eta < 1$ is the learning rate. This process is repeated until some convergence criterion is satisfied.

The loss function is typically the L_2 distance of the network output and the ground truth label, and some regularization, i.e. given ground truth label $\mathbf{y}$, network output $\mathbf{f}_{\boldsymbol{\theta}}$, and regularization strength λ, we have

$$L(\mathbf{x}, \mathbf{y}) = |\mathbf{y} - \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x})|_2 + \lambda|\boldsymbol{\theta}|_2,$$

where $|\cdot|_2$ is the Euclidean distance (i.e. L_2 distance.)

A common way to extend SGD is through *mini-batch* processing, where a batch of samples are processed, and the gradient update is computed by the mean of each sample's gradient. A further improvement is momentum learning, where consecutive updates in the same direction contribute to "momentum" in that direction.

C. Network Parallel Training

Network Parallel Training (NPT) [3] is sometimes referred to as Unit Parallelization. In NPT, one divides the network units within each layer across different processes. In its simplest form, each process is responsible for calculating activation of a single unit in each layer [6]. In a more practical setup the units are divided in a linearly load balanced manner. See figure 1 for an example.

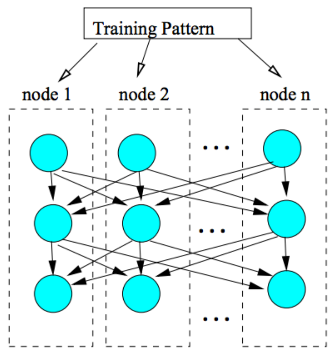

Figure 1. Illustration of the unit division in Network Parallel Training [3].

In our approach, the strategy is to compute the matrix-matrix multiplication XW_i for each layer in parallel using Cannon's algorithm for matrix multiplication, then compute the activation output for that unit. The shifting of the local matrices within Cannon's algorithm as defined in algorithm 1 allows for the partial products required for the calculation of activations in each unit to be available within the corresponding process.

D. Cannon's Algorithm

Cannon's algorithm is a popular parallelization strategy for matrix multiplication[7]. The algorithm efficiently computes matrix multiplications by shifting rows and columns of the candidate matrices between processes based on a defined set of rules. Algorithmm 1 shows Cannon's algorithm of two candidate matrices A $(M\times N)$ and B (N$\times K)$ within a grid of $p\times q$ processes. In the context of Backpropagation, Cannon's algorithm effectively allows communication of weights between nodes that are placed within different processes.

Algorithm 1: Cannon's Matrix Multiplication

```
Data: Number of parts $N$, sub-matrices $A, B$
Result: Local part $C$ of global matrix multiplication
begin
    Shift rows of $A$ such that the $m$th row of $A$ shifts
      $m-1$ positions to the left
    Shift columns of $B$ such that the $n$th column of $B$
      shifts $n-1$ positions to upwards
    for $i \in \{1, 2, \ldots, N\}$ do
        $C \longleftarrow C + AB$
        Shift rows of $A$ one process to the left, and
          columns of $B$ one process upwards
    end
    Undo initial shifts in $A$ and $B$
end
```

E. *Implementation*

We implement our Neural Network in C using the Message Passing Interface (MPI) [8]. We use the MPI_Cart_create function to create a virtual Cartesian grid topology that enables parallelization as depicted in fig. 1.

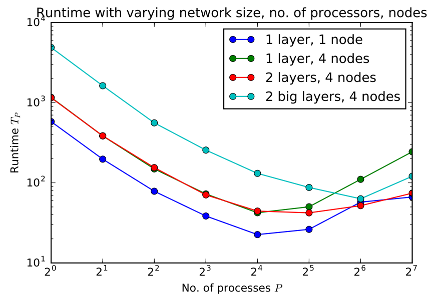

Figure 2. Plot of the runtimes T_P.

III. Experiments

A. *Training Data*

The data was created artificially by imposing a linear relationship between the inputs X_{ij} and the outputs Y_{ik}. The inputs are created by sampling random numbers uniformly between 0 and 200, with the bias input set to 1. The output vector is created using

$$Y_{ik} = ((7(k+1) \bmod 1000) + (3k \bmod 100))X_{ik}.$$

Interestingly, since 7, 3 and 1000 are all pairwise coprimes, this forms a basis for all linear functions $y = kx + m$ with parameters $k, m \in [0, 1000)$.

B. *Experiment Setup*

In order to evaluate the performance of our NPT algorithm we created the following networks of different sizes and complexity:

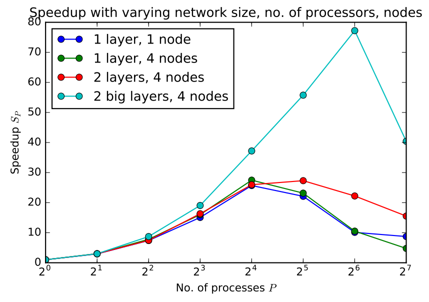

Figure 3. Plot of speedup S_P. We define the speedup at P processes as $S_P = \frac{T_S^*}{T_P}$ where T_P is the execution time of our algorithm with P processes. T_S^* is the "fastest known sequential version", since we do not know of any better algorithm, we let $T_S^* = T_1$.

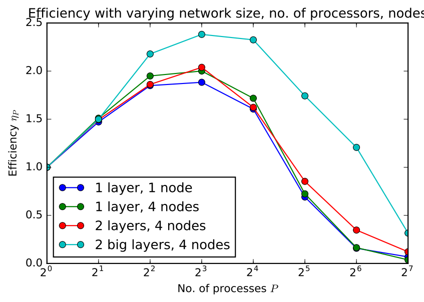

Figure 4. Plot of the efficiency $\eta_P = \frac{S_P}{P}$.

1) A One hidden layer network with the number of input and output units $q = 128$
2) A Two hidden layer network with the number of input and output units $q = 128$
3) A Two hidden layer network with the number of input and output units $q = 256$ - we refer to this as the "big" network.

We used an MPI barrier to synchronize before starting the training routine, and immediately stored the wall-clock time. The same was done after training completed. The runtime data is presented as relative runtimes in table I.

IV. Experiment Results

The networks above were run on an HPC cluster with an increasing number of processors. The resulting runtimes in seconds are shown in figure 2 together with the relative performance statistics in table I. It can be seen that for the single layer networks the runtimes decrease as processes increase until we obtain the optimal point with 16 processes; after this point communication cost becomes prohibitive resulting in increasing runtimes. As the number of nodes are increased from 1 to 4, the runtime also increases which might imply the additional cost of redundant communication and startup related to the unused nodes.

Increasing the number of layers l from 1 to 2 increases the optimal number of processes from 16 to 32 which gives empirical evidence that the runtimes are linear in the number of layers l .The runtimes for the "big" network increase quadratically relative to the two layered network on the same number of nodes Thus the optimal number of processes for the "big layer" network is 64.

It is rather surprising that the runtimes of the two layered network are lower than those of the single layered after 32 processes by a significant margin on the same number of nodes. This is a point that will require further investigation.

The speedup curve in figure 3 shows similar results as in the runtime plot with exactly the same optimal process parameters. The efficiency plot in figure 4 shows that it is more processor efficient to have 8 processors - above this the overall benefit (increase in speedup) of having additional processes decreases.

P	**1 layer, 1 node**	**2 layer, 1 node**	**2 layer, 4 node**	**2 big layers, 4 node**
1	25.69	51.46	51.00	215.88
2	8.73	17.04	17.07	71.98
4	3.47	6.60	6.85	24.78
8	1.71	3.21	3.13	11.33
16	1.00	1.87	1.96	5.80
32	1.16	2.22	1.87	3.87
64	2.54	4.89	2.30	2.80
128	2.94	10.80	3.28	5.34

Table I
RELATIVE PERFORMANCE OF THE DIFFERENT WORKLOADS AS MEASURED IN WALL-CLOCK TIME WITH THE BASELINE AS A 1 LAYER NETWORK WITH ON SINGLE NODE WITH 16 PROCESSES.

An important observation from the efficiency plots is that the processor efficiency exceeds 1 in certain regions - this implies that the seedup is superlinear [9]. Thus meaning - adding additional processes until a certain point actually increases the speedup possibly due to reduction in RAM access times.

V. CONCLUSIONS AND REMARKS

An implementation of Network Parallel Training using Cannon's algorithm was proposed. Running times, Speedup and Efficiency were evaluated on networks of varying complexity. Results show that when using this strategy significant speedup is obtained. This effect is even more pronounced for more complex networks. We have also showed that there are regions where such speedup is super-linear thus increasing the efficiency of resource utilization.

It would be interesting to explore different means of parallelization other than a per-unit based split of the weight matrices. The topologies tested here were of relatively limited size. Parallelized neural networks are mostly necessary once the networks are of unusually large size, otherwise communication time is going to far outweigh any benefits of parallel execution due to the many data dependencies.

The data model used was a simple linear function depending only on one variable. While it was not relevant for the scope of this work, implementing a more sophisticated means of data generation would be an interesting next step.

Another aspect to explore is how the local matrix multiplication can be improved. Perhaps offloading the matrix multiplication to a graphics processing unit would be beneficial.

ACKNOWLEDGEMENTS

The computations in this work were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at PDC Centre for High Performance Computing (PDC-HPC).

REFERENCES

[1] Marwala T. Economic Modeling Using Artificial Intelligence Methods. Springer Publishing Company, Incorporated; 2013.

[2] Farber P, Asanovic K. Parallel neural network training on Multi-Spert. In: Algorithms and Architectures for Parallel Processing, 1997. ICAPP 97., 1997 3rd International Conference on; 1997. p. 659–666.

[3] Dahl G, McAvinney A, Newhall T. Parallelizing Neural Network Training for Cluster Systems. In: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks. PDCN '08. Anaheim, CA, USA: ACTA Press; 2008. p. 220–225.

[4] Suri NNRR, Deodhare D, Nagabhushan P. Parallel Levenberg-Marquardt-based Neural Network Training on Linux Clusters - A Case Study. In: Linux Clusters, ICVGIP 2002, 3rd Indian Conference on Computer Vision, Graphics and Image Processing, Ahmadabad; 2002. .

[5] Long LN, Gupta A. Scalable Massively Parallel Artificial Neural Networks. JACIC. 2008;5:3–15.

[6] Pethick M, Liddle M, Werstein P, Huang Z. Parallelization of a backpropagation neural network on a cluster computer. In: International conference on parallel and distributed computing and systems (PDCS 2003); 2003. .

[7] Cannon LE. A Cellular Computer to Implement the Kalman Filter Algorithm. Bozeman, MT, USA; 1969. AAI7010025.

[8] Snir M, Otto SW, Walker DW, Dongarra J, Huss-Lederman S. MPI: The Complete Reference. Cambridge, MA, USA: MIT Press; 1995.

[9] Wilkinson B, Allen M. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.; 1999.